final NAG3-2177

Introduction

This report describes the research effort to demonstrate the integration of a data sharing technology, *Ring Buffered Network Bus,* in development by Dryden Flight Research Center, with an engine simulation application, *the Java Gas Turbine Simulator,* in development at the University of Toledo under a grant from the Glenn Research Center. The objective of this task was to examine the application of the RBNB technologies as a key component in the data sharing, health monitoring, and system wide modeling elements of the NASA Aviation Safety Program (AvSP) [Golding, 1997].

System-wide monitoring and modeling of aircraft and air safety systems will require access to all data sources which are relative factors when monitoring or modeling the national airspace such as radar, weather, aircraft performance, engine performance, schedule and planning, airport configuration, flight operations, etc. The data sharing portion of the overall AvSP program is responsible for providing the hardware and software architecture to access and distribute data, including real-time flight operations data, among all of the AvSP elements. The integration of an engine code capable of numerically "flying" through recorded flight paths and weather data using a software tool that allows for distributed access of data to this engine code demonstrates initial steps toward building a system capable of monitoring and modeling the National Airspace.

Overview

The current prototype allows users to perform a gas turbine simulation, store the execution results on the Ring Buffer Network Bus (RBNB), and access and display the results from distributed machines across a network. The topology of the integrated system is shown in Figure 1. There are three main components to the system: the Java Gas Turbine Simulator, the Ring Buffer Network Bus, and a Java-based Plotting application/applet.

- The Java Gas Turbine Simulator (JGTS) [Reed and Afjeh, 1997] is an object-oriented, interactive, graphical, numerical gas turbine simulator written entirely in Java™ [Arnold and Gosling, 1996]. It couples a graphical user interface, developed using the Java Abstract Window Toolkit, and a transient, space- averaged, aero-thermodynamic gas turbine analysis method, to provide an environment which allows the quick, efficient construction and analysis of arbitrary gas turbine systems. The combined package provides analytical, graphical and data management tools which allow the user to construct and control dynamic gas turbine simulations by manipulating graphical objects on the computer display screen. The Java language and environment permit user to easily access and run the simulator from a variety of heterogeneous computer platforms including PC's, Macintosh, and Unix machines.

- The RBNB DataTurbine is a software network data server that provides widely distributed users simultaneous access to real-time information [Freudinger and Miller, 1997]. The RBNB acts as an intermediary between dissimilar data monitoring and analysis algorithms and can be treated as a "black box" to which data is sent and received. It uses Java and standard Internet protocols.

- The Java-based plotting application/applet is a stand-alone version of the graphing tool available in the JGTS. The application/applet is used to display the results of the gas turbine simulation.

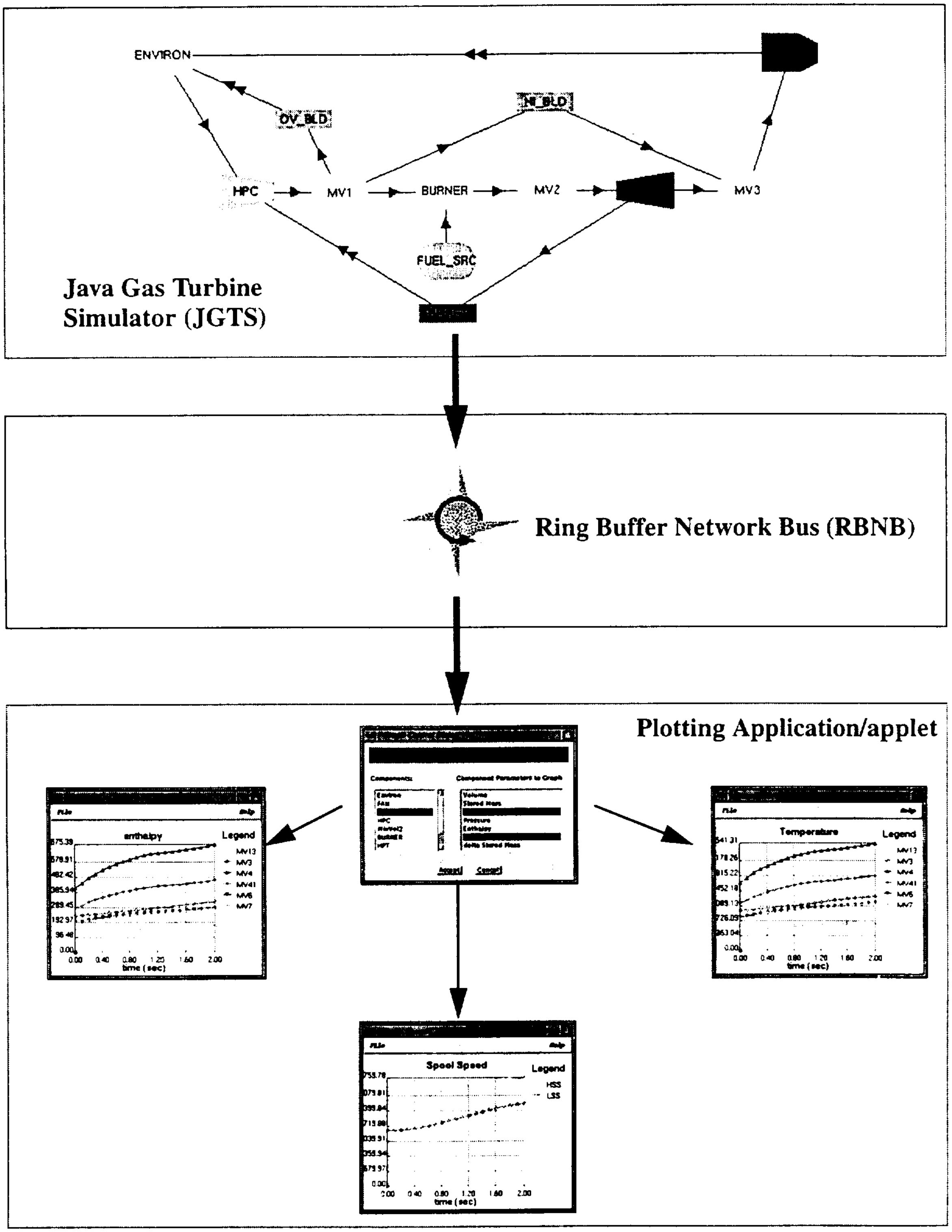


Figure 1: Topology of JGTS-RBNB data sharing system.

How it works

Gas turbine models are developed and simulated by the user in the **JGTS**. These models are represented as component objects which are instantiated and connected by the user to form an engine model (see Figure 2). Once an engine model is constructed, model-specific parameters (e.g., temperature, mass flow rate, etc.) are entered into each component. The user can select an appropriate numerical method from a library of numerical solvers to carry out the simulation. At each converged time-step during the simulation, the data describing the component operations in the engine (i.e., the various component parameters) are sent to the **RbnbWriteMgr** object. This singleton object [Gamma *et al*, 1995] provides a global point of access to the RBNB, and also serves as a temporary database for storing component parameters during the simulation. At the successful conclusion of the simulation, this data is encoded and transmitted across a network to the **RBNB Server** where it is stored. Users wishing to access the simulation results can use the Plotting application/applet from networked workstations to connect to the RBNB server. When started, the singleton **RbnbReadMgr** object accesses and decodes the simulation data previously stored on the RBNB server. This data is passed to the **RbnbPlotController** which stores the data in a

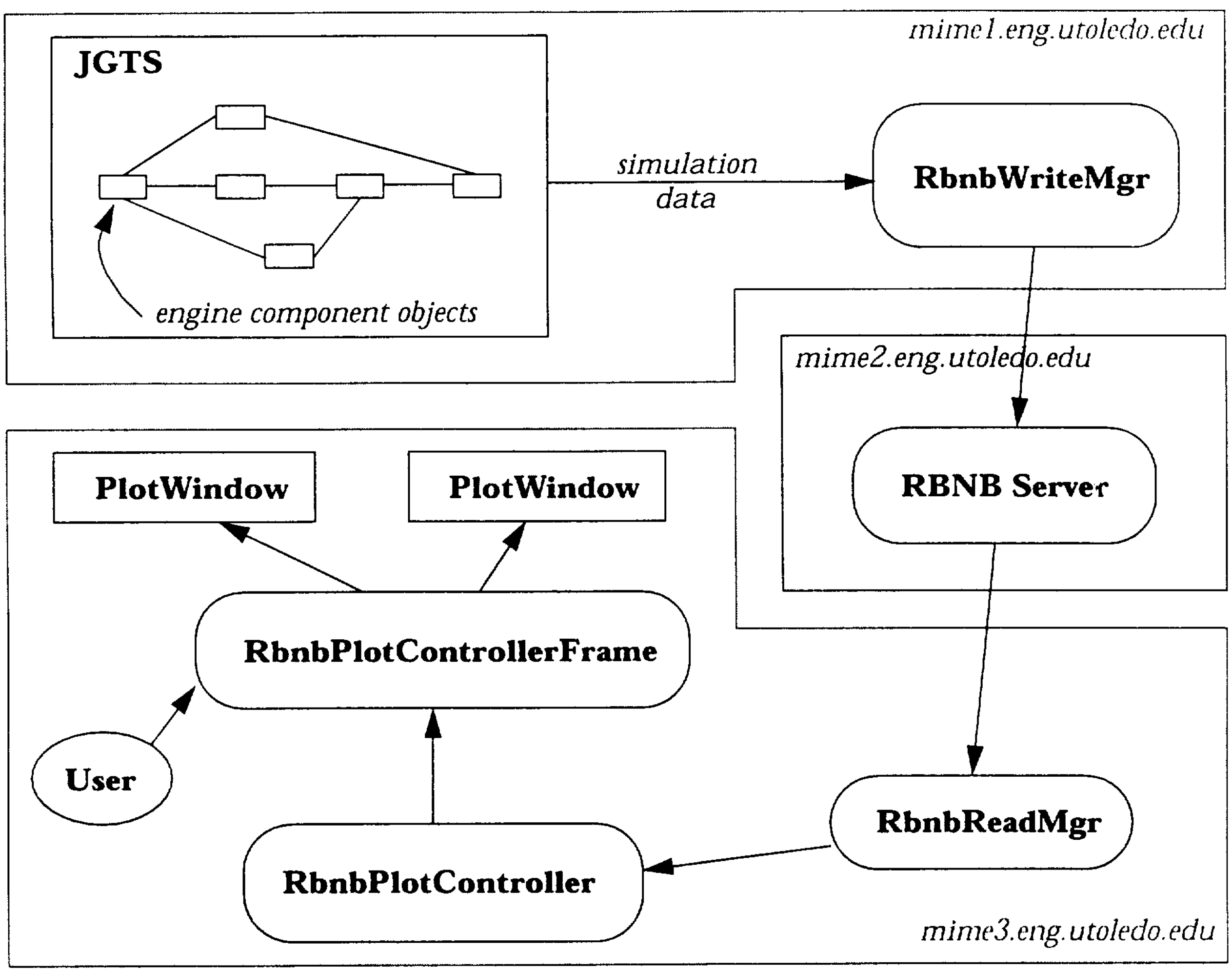


Figure 2: Interaction diagram of main components in data sharing system.

simple internal database, and builds the **RbnbPlotControllerFrame** user interface. The RbnbPlotControllerFrame provides graphical lists of the engine components in the simulation and their parameters. Selection of a parameter from the list creates a graphical **PlotWindow** of the parameter to be displayed. For example, Fig. 1 shows PlotWindows for enthalpy, temperature and spool speed parameters.

Simulation Execution

The first step in running the simulation is to start the RBNB Server. The RBNB Server is a Java application, and is started using the command:

```
java rbnbServer &
```

This starts up the Java Virtual Machine, executes the rbnbServer.class file, and places the server in the background so that it runs continuously. By default, the RBNB listens to TCP/IP port 3333. Other ports can be specified at startup using the *-s* command-line argument. For example,

```
java rbnbServer -s 7777 &
```

will instruct the RBNB to listen to TCP/IP Port 7777 for connecting applications.

Once the RbnbServer is running, the user can execute the JGTS engine simulation. It is assumed that a valid JGTS engine model is available; the prototype uses a model of the Pratt & Whitney F100 engine, which was developed previously [Reed and Afjeh, 1997]. The simulation is started from with the command:

```
java engine-model-class rbnb-location:port
```

The *engine-model-class* argument defines the Java class file containing the JGTS engine model. The *rbnb-location:port* argument is the TCP/IP address and port of the Rbnb Server where the simulation results will be stored. For example, if the JGTS engine model is contained in the file `jnpss.engine.NPSS` and the RBNB is running on `mime2.eng.utoledo.edu` using port 7777, then the user would start the simulation with the command:

```
java jnpss.engine.NPSS mime2.eng.utoledo.edu:7777
```

Simulation Initialization

When the engine simulation model is executed, the JGTS simulation system performs a series of initialization steps. One of these steps is the initialization of the `RbnbWriteMgr` singleton object. During this simulation initialization, the `NpssSystem` object (which is the controller in the JGTS system) sends the `initialize()` message to the `RbnbWriteMgr` singleton (see interaction diagram, Fig. 3). The `initialize()` method queries the `NpssSystem` object to determine the engine components in the system and uses this information to construct the `componentNames` Vector (see Table 1). As its name implies, `componentNames` is a collection of the names of the components currently in the engine model.

Table 1: componentNames Vector example

componentName1-->	ENVIRONMENT
componentName2-->	FAN
componentName3-->	MV13
componentName4-->	HPC
.	
.	
componentNameN-->	NOZZLE

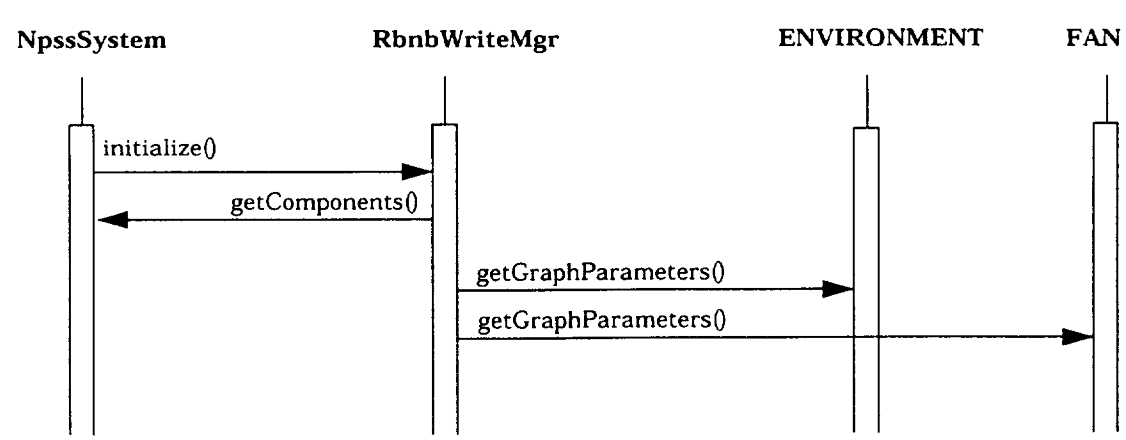


Figure 3: Object interaction diagram for initialization.

The initialize() method also sends the message getGraphParameters() to each component in the engine model to identify the parameters (e.g., pressure, temperature, mass flow rate, etc.) for each component. Using this information, RbnbWriteMgr creates the parameterNamesTable Hashtable to identify the names of the parameters for each component. The Hashtable key is the componentName, and the value is a Vector containing the names (Strings) of the parameters for that component. The names of the parameters are added to the Hashtable at this time. Note that each component type (e.g, Compressor, MixingVolume, Turbine, etc.), will have a specific number of parameters; these may not be equal. An example of the parameterNamesTable structure is shown in Table 2.

Table 2: RbnbWriteMgr parameterNamesTable example

ComponentName-->	ENVIRONMENT	FAN	HPC	MV13	...
paramValue1-->	enthalpy	enthalpy	enthalpy	volume	
paramValue2-->	pressure	pressure	pressure	pressure	
paramValue3-->	temperature	temperature	temperature	temperature	
paramValue4-->	altitude	massFlowRate	massFlowRate	deltaMass	
paramValue5-->	specificHeat	specificHeat	specificHeat	specificHeat	
.	↑	↑	↑	↑	↑
.	↓	↓	↓	↓	↓
paramValueN-->	machNumber	energyFlux	energyFlux	energyFlux	

Also created in RbnbWriteMgr initialize() method is the paramValuesTable Hashtable. In this Hashtable, the key is the component name, the value is a Vector containing the time-step and parameter value sets (see below). At this time, the Vectors are empty.

Data Generation

During a transient engine simulation, the numerical solver in the JGTS attempts to force the engine model to convergence at each time step. When this occurs, the solver invokes the NpssSystem object's updateGraphs() method (see Figure 4). This method gets the single instance of the RbnbWriteMgr class, and invokes it's updateData() method, which invokes it's own getDataFromNpssSystem() method. In this method, each component in the engine model is sent the getGraphParameters() message which returns a Hashtable containing the names of the parameters and their current values. This data is then stored in the paramValuesTable Hashtable in RbnbWriteMgr.

The paramValuesTable Hashtable key is a String identifying the component name and the value is a Vector. The Vector structure is shown in Table 3. As the simulation progresses, parameter values at each time step are added to each Vector corresponding to the appropriate component name. At the conclusion of the simulation, the paramValuesTable will be completely filled with parameter values at each time step for each component in the simulation (see Table 4, for example).

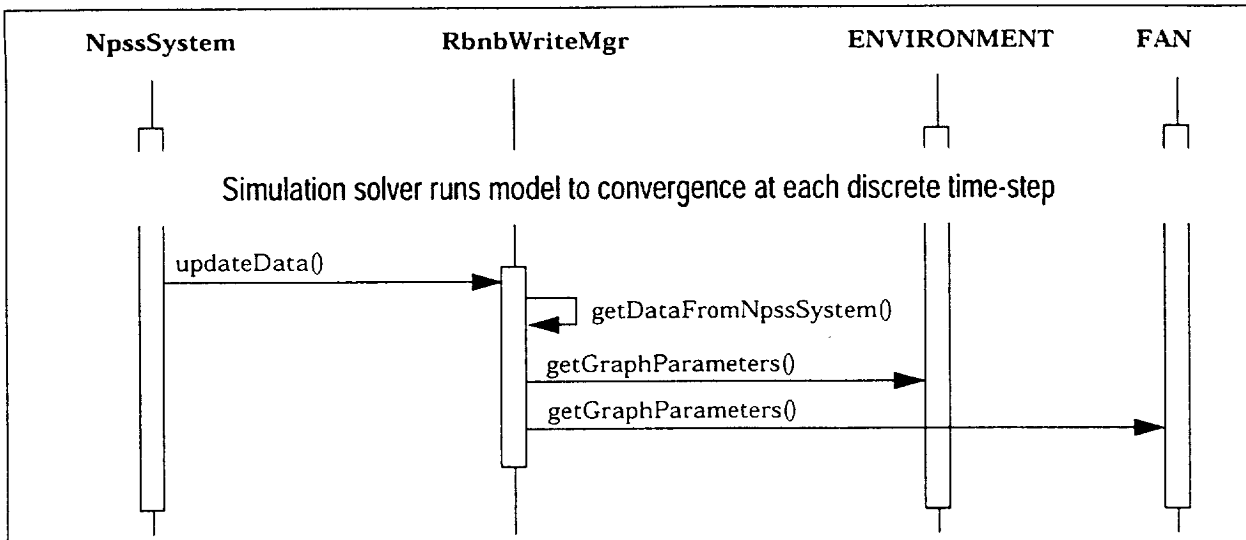


Figure 4: Object interaction diagram for data generation.

Sending the Data to the RBNB

If the transient simulation completes successfully, the `NpssSytem` object calls `RbnbWriteMgr.updateRBNB()` which calls its own `putDataOnRbnb()` method (see Figure 5). This method establishes a connection to the RBNB, creates an RBNB `Map` in which to store the data, and proceeds to loop through each of the objects in the `componentNames` Vector. For each entry in the Vector, an RBNB `Channel` object is created to hold the component's simulation data. A Channel object stores data in the form of a `byte` array. To convert the `String`, `int`, and `double` values into bytes, and put them into an array. `RbnbWriteMgr` uses the `encodeDataArray()` to create the Channel's byte data array. Here is the method code (without exception handling):

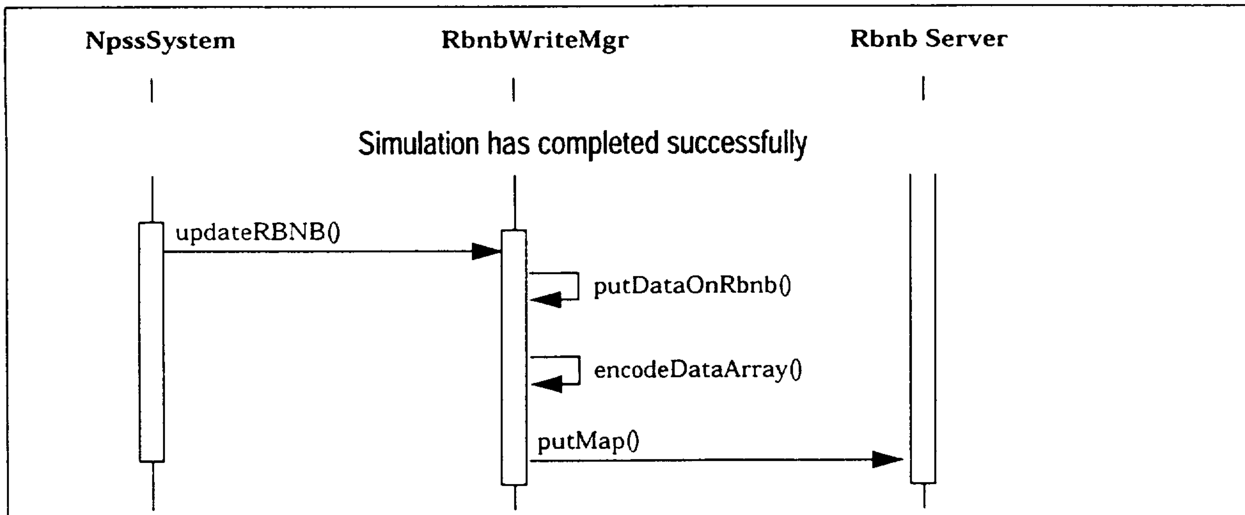


Figure 5: Object interaction diagram for sending data to the RBNB.

Table 3: RbnbWriteMgr paramValuesTable description

componentName	String	The component name.
timeValue1	Double	The first simulation time-step value for the following parameter values.
paramValue1	Double	The first component parameter's (*paramName1*) value.
paramValue2	Double	The second component parameter's (*paramName2*) value.
.		
.		
paramValueN	Double	The "*n*"th component parameter's (*paramNameN*) value.
timeValue2	Double	The second simulation time-step value for the following parameter values.
paramValue1	Double	The first component parameter's (*paramName1*) value.
paramValue2	Double	The second component parameter's (*paramName2*) value.
.		
.		
paramValueN	Double	The "*n*"th component parameter's (*paramNameN*) value.
↕		
timeValueM	Double	The "*m*"th simulation time-step value for the following parameter values.
paramValue1	Double	The first component parameter's (*paramName1*) value.
paramValue2		The second component parameter's (*paramName2*) value.
.		
.		
paramValueN	Double	The "*n*"th component parameter's (*paramNameN*) value.

Table 4: RbnbWriteMgr paramValuesTable example

	Compressor	Fan	MV13	HPC	MV3
timeValue1 -->	0.1	0.1	0.1	0.1	0.1
paramValue1 -->	100.0	55.3	453453.0	454.0999	555.33
paramValue2 -->	545.9	89890.0	33.0	23234.5	2323.5
.					
.					
paramValueN -->	5.6343	222.33	74545.3	8.3434	844343.3
timeValue2 -->	0.2	0.2	0.2	0.2	0.5
paramValue1 -->	101.3	56.7	453453.0	456.9933	567.3
paramValue2 -->	544.3	-22200.0	35.674	26423.67	2554.7
.					
.					
paramValueN -->	7.2323	266.343	88234.2	10.3343	723356.2
	↕	↕	↕	↕	↕
timeValueM -->	1.0	1.0	1.0	1.0	1.0
paramValue1 -->	236.3	89.4	234555.2	678.9643	10223.3
paramValue2 -->	6854.4	0.2323	653.3	2343.3	233.3
.					
.					
paramValueN -->	6.34343	299.44	93993.3	11.1112	7234.3

```
private byte[] encodeDataArray(String componentName) throws Exception {

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(baos);

      // Write component name
      oos.writeObject(componentName);

      // Write the number of parameters to data array
      Integer i1 = (Integer)numParametersTable.get(componentName);
      int numOfParameters = i1.intValue();
       oos.writeInt(numOfParameters);

      // Loop through v1 and copy the names of the parameters to the ObjectOutputStream
      Vector v1 = (Vector)paramNamesTable.get(componentName);
      for (int k = 0; k < v1.size(); k++) {
              String s = (String)v1.elementAt(k);
              oos.writeObject(s);
      }

      // Get the vector for the paramValuesTable and copy the parameter values
      // to the ObjectOutputStream
      Vector v2 = (Vector)paramValuesTable.get(componentName);
      for (int m = 0; m < v2.size(); m++) {
              Double d1 = (Double)v2.elementAt(m);
              double d2 = d1.doubleValue();
              oos.writeDouble(d2);
      }

      // Get contents of ObjectOutputStream as byte array
      oos.flush();
      byte[] data = baos.toByteArray();

      // Release the ObjectOutputStream resources
      oos.close();

      return data;
  }
```

The returned `byte` array is stored into the appropriate Channel object which is then registered with the RBNB Map object created previously. This process is repeated for each of the entries in the `componentNames` Vector, after which, the Map object is sent to the RBNB.

Data Channel Structure

For the prototype, we used a single RBNB data Channel object to store data for each component in the engine model. The data format for the channel is shown in Table 5. The data is arranged as a vector of Java `String`, `int` and `double` entries.

<u>component Name</u>

The default `Channel` data format, shown in Table 5, starts with a String representing the Component's name. This `String` is also used as the name of the RBNB `Channel`, so that the names of the components can be identified from the RBNB `Channel Map`.

Table 5: Format of data in Channel

component Name	String	The component name.
numOfParams	int	The number of component parameters.
paramName1	String	The name of the first component parameter
paramName2	String	The name of the second component parameter.
.	String	
.	String	
paramNameN	String	The name of the "*n*"th component parameter.
timeValue1	double	The first simulation time-step value for the following parameter values.
paramValue1	double	The first component parameter's (*paramName1*) value.
paramValue2	double	The second component parameter's (*paramName2*) value.
.	double	
.	double	
paramValueN	double	The "*n*"th component parameter's (*paramNameN*) value.
timeValue2	double	The second simulation time-step value for the following parameter values.
paramValue1	double	The first component parameter's (*paramName1*) value.
paramValue2	double	The second component parameter's (*paramName2*) value.
.	double	
.	double	
paramValueN	double	The "*n*"th component parameter's (*paramNameN*) value.
.	.	.
.	.	.
.	.	.
timeValueM	double	The "*m*"th simulation time-step value for the following parameter values.
paramValue1	double	The first component parameter's (*paramName1*) value.
paramValue2	double	The second component parameter's (*paramName2*) value.
.	double	
.	double	
paramValueN	double	The "*n*"th component parameter's (*paramNameN*) value.

numOfParams
The next entry in the Channel is the number of parameters which this component is storing. Parameters, such as temperature, pressure, mass flow rate, are varied and dependent on the type of engine model being used. Because each component may have different numbers of parameters, this value is needed to for encoding/decoding the data. The list of component parameters are automatically obtained by RbnbWriteMgr before the simulation begins.

paramName1, paramName2, ..., paramNameN
Each parameter's name is defined by a String which is added as a separate entry to the data channel.

time, paramValue1, paramValue2, ..., paramValueN
After the parameter names are added, the data results form the simulation are added. The data is defined by a time value and then values for each parameter at that time step. First the time-step term is added, then the values for each parameter at that time-step is added. Note, the order of parameter value addition is the same as the order of parameter names. Additional time-step and parameter values entries are added until the data is exhausted.

Extracting and Plotting Simulation Data

Multiple users residing at different locations on a network may access and display the simulation results stored on the RBNB using the prototype systems's Plotting tool. The Plotting tool is designed to be run either as a Java applet or a Java application. In the case of an applet, the tool is run in the context of a Java-enabled browser such as Netscape Navigator™ or HotJava. The applet is embedded in an HTML page which is loaded and executed automatically. When run as a application, the Plotting tool is started from the command line with the command:

```
java RbnbPlotController rbnb-location:port
```

Where the *rbnb-location:port* argument is the TCP/IP address and port of the Rbnb Server holding the simulation results. For example, to access and display the results stored in the RBNB Server running on mime2.eng.utoledo.edu using port 7777, we would have:

```
java RbnbPlotController mime2.eng.utoledo.edu:7777
```

When the Plotting tool is started, the RbnbPlotController class performs two functions: 1) it initiates the loading and storing of simulation results from the RBNB Serve, and 2) it creates an instance of RbnbPlotControllerFrame, which acts as the graphical user interface for interacting with the user.

Data Extraction

The data extraction process is essentially the reverse of the data encoding process described above. After getting the singleton RbnbReadMgr object, RbnbPlotController invokes its getDataFromRbnb() method to start the data extraction process. RbnbReadMgr calls

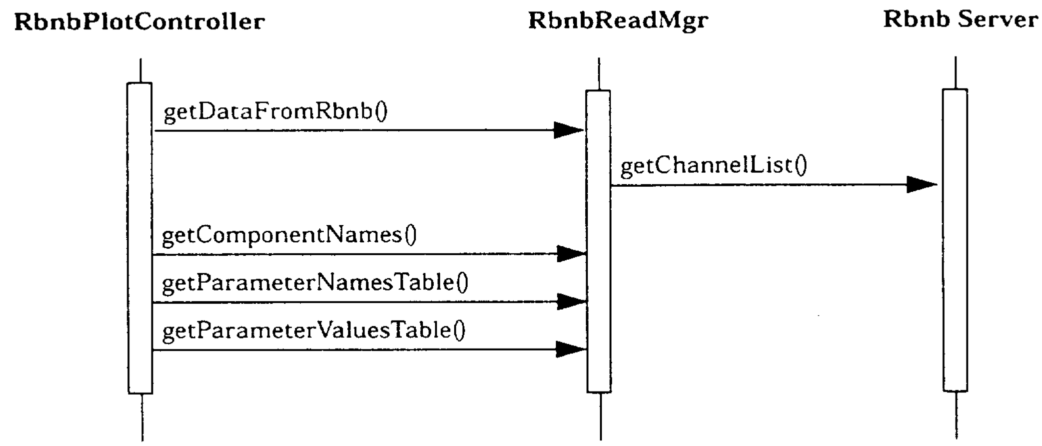


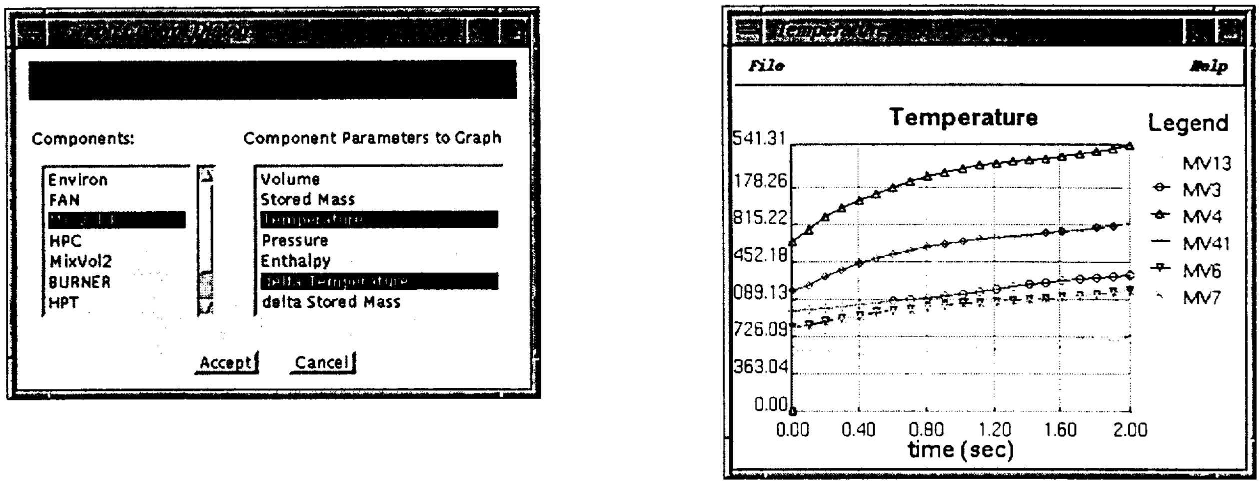
Figure 6: Object interaction diagram for retrieving data from the RBNB.

`getChannelList()` which returns a array of all `Channels` currently on the RBNB. `RbnbReadMgr` then iterates through each Channel to obtain a Map for the simulation results. Identification of the correct simulation data is specified by a beginning and ending `TimeStamp`. This `TimeStamp` value was created when the data was stored. Currently users can only retrieve the first data set. In the future, we might create a mechanism to examine all simulation sets currently stored on the RBNB. For the prototype, a single data set was sufficient to build and test the system.

The list of `Channel` objects in the `Map` is obtained and iterated over so that each `Channel's` `byte` array can be retrieved. The `String` objects and primitive elements (`int` and `double`) are extracted in exactly the same order as was used by the `RbnbWriteMgr` to encode the data. During this "decoding" process, the component names and parameter values are extracted and temporarily stored in the `RbnbReadMgr`. Storage of the data is in the same form as described for the `RbnbWriteMgr`: `componentNames` Vector, `parameterNamesTable` Hashtable, and `paramValuesTable` Hashtable. Once the data has been extracted, the `RbnbPlotController` calls the `RbnbReadMgr` object to get the `componentNames` Vector, the `parameterNamesTable` Hashtable, and `paramValuesTable` Hashtable. `RbnbPlotController` then adds the retrieved data to an internal database which supports the operations of the `RbnbPlotControllerFrame` graphical user interface.

Data Display

The RbnbPlotControllerFrame provides the user with a graphical user interface (GUI) with which to select the engine model simulation results. The GUI is shown in Figure 7(a). The GUI is comprised of two List components. The left-hand List displays the name of each of the engine components which were present in the engine simulation. The right-hand List displays the name of the parameters to be graphed for the selected engine component. This List allows multiple selections, allowing the user to select to graph any or all of the component's parameters. For example, in Figure 7(a), the `Temperature` and `deltaTemperature` parameters for `MixVol1` component have been selected.



(a) (b)

Figure 7: (a) Plotting tool graphical user interface; **(b)** Plot Window

Selection of a parameter automatically displays a `PlotWindow` for that parameter (see Figure 7(b)). Selection of the Temperature parameter for other components will cause their values to be plotted in the same PlotWindow.

Summary

This report describes the research effort to demonstrate the integration of a data sharing technology, Ring Buffered Network Bus, in development by Dryden Flight Research Center, with an engine simulation application, the Java Gas Turbine Simulator, in development at the University of Toledo under a grant from the Glenn Research Center.

The initial objectives of the project have been met. We have obtained and reviewed the RBNB software and documentation and established a basic understanding of the system. Furthermore, we have demonstrated the application of the RBNB to data sharing by integrating the RBNB with the Java Gas Turbine Simulator so that simulation data generated by the JGTS could be stored on the RBNB and later retrieved and displayed by remote users using a Java Plotting tool.

This work required the investigation, design and development of a set of Java interfaces, abstract classes and concrete classes, to define the interface between JGTS and RBNB. For expediency these interfaces were designed based on current component object models in only the JGTS. However, the design used is sufficiently general so that connections to other engine simulation systems, such as NCP, can be made without re-factoring the system. This was accomplished by the use of design patterns, such as the Singleton pattern, and advanced object-oriented framework design techniques. As a result, it should be possible to "plug" in new functionality without redesign. This should also limit effects from changes in the RBNB software as it evolves.

References

Arnold, K. and Gosling, J., 1996, "The Java Programming Language," Addison Wesley Publishing Company, Inc., Reading, MA.

Freudinger, L.C., and Miller, M.J., 1997, "Middleware Technology for Aircraft Health Monitoring for Global Aviation Safety".

Gamma, E., Helm, R, Johnson, R., and Vlissides, J., 1995, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley Publishing Company, Inc., Reading, MA.

Golding, D. S., "The Three Pillars of Success for Aviation and Space Transportation in the 21st Century," a speech before the Aero Club, AIAA and NAC, March 20, 1997. Available at http://www.hq.nasa.gov/office/aero/oastthp/brochure/brochure.htm

Reed, J. A, and Afjeh, A.A, 1997, "A Java-based Interactive Graphical Gas Turbine Propulsion System Simulator," AIAA paper 97-0233, 35th Aerospace Sciences Meeting and Exhibit, Reno NV.